REPORT DOCUMENTATION PAGE

Form Approved
OPM No.

(2)

AD-A252 501

1. AGENCY USE (Leave	3. REPORT TYPE AND DATES Final: 30 Apr 1991 to 02 Jun 1993
4. TITLE AND Validation Summary Report: Alsys, AlsyCOMP_061 Version 1.83, DECstation 3100 under ULTRIX Version 4.2 (Host) to STAR MVP board (R3000/R3010)(Target), 920429I1.11251	5. FUNDING
6. IABG-AVF Ottobrunn, Federal Republic of Germany	
7. PERFORMING ORGANIZATION NAME(S) AND IABG-AVF, Industrieanlagen-Betriebsgeselschaft Dept. SZT/ Einsteinstrasse 20 D-8012 Ottobrunn FEDERAL REPUBLIC OF GERMANY	8. PERFORMING ORGANIZATION IABG-VSR 107
9. SPONSORING/MONITORING AGENCY NAME(S) AND Ada Joint Program Office United States Department of Defense Pentagon, Rm 3E114 Washington, D.C. 20301-3081	10. SPONSORING/MONITORING AGENCY
11. SUPPLEMENTARY	
12a. DISTRIBUTION/AVAILABILITY Approved for public release; distribution unlimited.	12b. DISTRIBUTION

13. (Maximum 200

Alsys, AlsyCOMP_061 Version 1.83, DECstation 3100 under ULTRIX Version 4.2 (Host) to STAR MVP board (R3000/R3010)(Target), ACVC 1.11.

92-17198

14. SUBJECT Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A,	15. NUMBER OF 16. PRICE

17. SECURITY CLASSIFICATION UNCLASSIFIED	18. SECURITY UNCLASSIFED	19. SECURITY CLASSIFICATION UNCLASSIFIED	20. LIMITATION OF

NSN

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std.

AVF Control Number: IABG-VSR 107
30 April 1991

Ada COMPILER VALIDATION SUMMARY REPORT:

Certificate Number: 920429I1.11251
Alsys
AlsyCOMP_061 Version 1.83
DECstation 3100 under ULTRIX Version 4.2 Host
STAR MVP board (R3000/R3010) Target

Prepared By:
IABG mbH, Abt. ITE
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 92-04-29.

Compiler Name and Version:	**AlsyCOMP_061 Version 1.83**
Host Computer System:	**DECstation 3100 under ULTRIX Version 4.2**
Target Computer System:	**Lockheed Sanders STAR MVP board (R3000/R3010) Target (bare machine)**

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate #920429I1.11251 is awarded to Alsys. This certificate expires 24 months after ANSI approval of MIL-STD 1815B.

This report has been reviewed and is approved.

[illegible]

IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany

[illegible] [illegible]

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

[illegible]

for Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Declaration of Conformance

Customer:	Alsys GmbH & Co. KG
Certificate Awardee:	Alsys
Ada Validation Facility:	IABG mbH, Germany
ACVC Version:	1.11

Ada Implementation:

AlsyCOMP_061 Version 1.83

Host Computer System:	DECstation 3100 under ULTRIX Version 4.2
Target Computer System:	Lockheed Sanders STAR MVP board (R3000/R3010) (bare machine)

Declaration:

I, the undersigned, declare that I have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.

[illegible]
Customer Signature

30.4.92
Date

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Computer and Software Engineering Division
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311-1772

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 02 August 1991.

E28005C	B28006C	C32203A	C34006D	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	B41308B	C43004A
C45114A	C45346A	C45612A	C45612B	C45612C	C45651A
C46022A	B49008A	B49008B	A74006A	C74308A	B83022B
B83022H	B83025B	B83025D	B83026B	C83026A	C83041A
B85001L	C86001F	C94021A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
AD7206A	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests) (*)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

(*) C24113W..Y (3 tests) contain lines of length greater than 255 characters which are not supported by this implementation.

The following 20 tests check for the predefined type LONG_INTEGER; for this implementation, there is no such type:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45613C	C45614C	C45631C	C45632C	B52004D
C55B07A	B55B09C	B86001W	C86006C	CD7101F

C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT; for this implementation, there is no such type.

C41401A checks that CONSTRAINT_ERROR is raised upon the evaluation of various attribute prefixes; this implementation derives the attribute values from the subtype of the prefix at compilation time, and thus does not evaluate the prefix or raise the exception. (See Section 2.3.)

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, MACHINE_OVERFLOWS is TRUE.

B86001Y uses the name of a predefined fixed-point type other than type DURATION; for this implementation, there is no such type.

C96005B uses values of type DURATION's base type that are outside the range of type DURATION; for this implementation, the ranges are the same.

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

CD2B15B checks that STORAGE_ERROR is raised when the storage size specified for a collection is too small to hold a single value of the designated type; this implementation allocates more space than was specified by the length clause, as allowed by AI-00558.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions; this implementation provides no package MACHINE_CODE.

The following 264 tests check operations on sequential, text, and direct access files; this implementation does not support external files:

CE2102A..C (3)	CE2102G..H (2)	CE2102K	CE2102N..Y (12)
CE2103C..D (2)	CE2104A..D (4)	CE2105A..B (2)	CE2106A..B (2)
CE2107A..H (8)	CE2107L	CE2108A..H (8)	CE2109A..C (3)
CE2110A..D (4)	CE2111A..I (9)	CE2115A..B (2)	CE2120A..B (2)
CE2201A..C (3)	EE2201D..E (2)	CE2201F..N (9)	CE2203A
CE2204A..D (4)	CE2205A	CE2206A	CE2208I
CE2401A..C (3)	EE2401D	CE2401E..F (2)	EE2401G
CE2401H..L (5)	CE2403A	CE2404A..B (2)	CE2405B
CE2406A	CE2407A..B (2)	CE2408A..B (2)	CE2409A..B (2)
CE2410A..B (2)	CE2411A	CE3102A..C (3)	CE3102F..H (3)
CE3102J..K (2)	CE3103A	CE3104A..C (3)	CE3106A..B (2)
CE3107B	CE3108A..B (2)	CE3109A	CE3110A
CE3111A..B (2)	CE3111D..E (2)	CE3112A..D (4)	CE3114A..B (2)
CE3115A	CE3119A	EE3203A	EE3204A
CE3207A	CE3208A	CE3301A	EE3301B
CE3302A	CE3304A	CE3305A	CE3401A
CE3402A	EE3402B	CE3402C..D (2)	CE3403A..C (3)
CE3403E..F (2)	CE3404B..D (3)	CE3405A	EE3405B
CE3405C..D (2)	CE3406A..D (4)	CE3407A..C (3)	CE3408A..C (3)
CE3409A	CE3409C..E (3)	EE3409F	CE3410A
CE3410C..E (3)	EE3410F	CE3411A	CE3411C
CE3412A	EE3412C	CE3413A..C (3)	CE3414A
CE3602A..D (4)	CE3603A	CE3604A..B (2)	CE3605A..E (5)
CE3606A..B (2)	CE3704A..F (6)	CE3704M..O (3)	CE3705A..E (5)
CE3706D	CE3706F..G (2)	CE3804A..P (16)	CE3805A..B (2)
CE3806A..B (2)	CE3806D..E (2)	CE3806G..H (2)	CE3904A..B (2)
CE3905A..C (3)	CE3905L	CE3906A..C (3)	CE3906E..F (2)

CE2103A, CE2103B, and CE3107A expect that NAME_ERROR is raised when an attempt is made to create a file with an illegal name; this implementation does not support the creation of external files and so raises USE_ERROR. (See section 2.3.)

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 22 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B24009A	B29001A	B38003A	B38009A	B38009B
B91001H	BC2001D	BC2001E	BC3204B	BC3205B	BC3205D

C34007P and C34007S were graded passed by Evaluation Modification as directed by the AVO. These tests include a check that the evaluation of the selector "all" raises CONSTRAINT_ERROR when the value of the object is null. This implementation determines the result of the equality tests at lines 207 and 223, respectively, based on the subtype of the object; thus, the selector is not evaluated and no exception is raised, as allowed by LRM 11.6(7). The tests were graded passed given that their only output from Report.Failed was the message "NO EXCEPTION FOR NULL.ALL - 2".

C41401A was graded inapplicable by Evaluation Modification as directed by the AVO. This test checks that the evaluation of attribute prefixes that denote variables of an access type raises CONSTRAINT_ERROR when the value of the variable is null and the attribute is appropriate for an array or task type. This implementation derives the array attribute values from the subtype; thus, the prefix is not evaluated and no exception is raised, as allowed by LRM 11.6(7), for the checks at lines 77, 87, 97, 108, 121, 131, 141, 152, 165, & 175.

C64103A was graded passed by Evaluation Modification as directed by the AVO. This test checks that exceptions are raised when actual parameter values, which result from an explicit type conversion, do not belong to the formal parameter's base type. However this implementation recognizes that the formal parameter is not used within the procedure and therefore the type conversion (and subtype check) need not be made (as allowed by [Ada 83] 11.6.7) and the subsequent expected exception need not be raised. The AVO ruled that the implementation's behavior should be graded passed, given that Report.Failed was invoked only from procedure calls at lines 91 (invoking line 76) and 119 (invoking line 115), yielding the following output:

```
"EXCEPTION NOT RAISED BEFORE CALL -P2 (A)"
"EXCEPTION NOT RAISED BEFORE CALL -P3 (A)"
```

BC3204C..D and BC3205C..D (4 tests) were graded passed by Evaluation Modification as directed by the AVO. These tests are expected to produce compilation errors, but this implementation compiles the units without error; all errors are detected at link time. This behavior is allowed by AI-00256, as the units are illegal only with respect to units that they do not depend on.

CE2103A, CE2103B, and CE3107A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests abort with an unhandled exception when USE_ERROR is raised on the attempt to create an external file. This is acceptable behavior because this implementation does not support external files (cf. AI-00332).

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For technical and sales information about this Ada implementation system in Germany, see:

Alsys GmbH & Co. KG
Am Rüppurrer Schloß 7
W-7500 Karlsruhe 51
Germany
Tel. +49 721 883025

For technical and sales information about this Ada implementation system outside Germany, see:

Alsys Inc.
67 South Bedford Str.
Burlington MA
01803-5152
USA
Tel. +617 270 0030

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system -- if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

a) Total Number of Applicable Tests	3556	
b) Total Number of Withdrawn Tests	95	
c) Processed Inapplicable Tests	54	
d) Non-Processed I/O Tests	264	
e) Non-Processed Floating-Point Precision Tests	201	
f) Total Number of Inapplicable Tests	519	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

3.3 TEST EXECUTION

A Magnetic Data Cartridge (TK 50) containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the cartridge were directly loaded onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system by a serial communications link (RS232), and run. The results were captured on the host computer system.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options.

Tests were compiled using the command

```
xada.c  -v 'file name'
```

and linked using the command

```
xada.link  -v -o 'file name' -d 'file name' -B 'file name' 'main
unit'.
```

The meaning of the options invoked explicitly for validation testing during this test is as follows

Compiler options :

-v	tells the Compiler to write additional messages onto the specified file,

Linker options :

-B	together with a parameter specifies the file containing the result of a previous incremental link, on which the final link is to be performed,
-o	together with a parameter specifies the name of the file which will contain the result of the final link,
-v	tells the implicitly invoked Completer to write additional messages onto the specified file,
-d	together with a parameter specifies the file which contains the linker directives.

Chapter B tests, the executable not applicable tests, and the executable tests of class E were compiled using the full listing option -l. For several tests, completer listings were added and concatenated using the option -L 'file name'. The completer is described in Appendix B, compilation system options, chapter 4.2 of the User Manual on page 40.

Test output, compiler and linker listings, and job logs were captured on a Magnetic Data Cartridge and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$MAX_IN_LEN	255 -- Value of V
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2_147_483_647
$DEFAULT_MEM_SIZE	2147483648
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	MIPS_BARE
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	SYSTEM.INTERRUPT_VECTOR(1)
$ENTRY_ADDRESS1	SYSTEM.INTERRUPT_VECTOR(2)
$ENTRY_ADDRESS2	SYSTEM.INTERRUPT_VECTOR(3)
$FIELD_LAST	512
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	0.0
$GREATER_THAN_DURATION_BASE_LAST	200_000.0
$GREATER_THAN_FLOAT_BASE_LAST	16#1.0#E+32
$GREATER_THAN_FLOAT_SAFE_LARGE	16#0.8#E+32
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	0.0
$HIGH_PRIORITY	15

$ILLEGAL_EXTERNAL_FILE_NAME1	file1
$ILLEGAL_EXTERNAL_FILE_NAME2	file2
$INAPPROPRIATE_LINE_LENGTH	-1
$INAPPROPRIATE_PAGE_LENGTH	-1
$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.TST")
$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006D1.TST")
$INTEGER_FIRST	-2147483648
$INTEGER_LAST	2147483647
$INTEGER_LAST_PLUS_1	2147483648
$INTERFACE_LANGUAGE	ASSEMBLER
$LESS_THAN_DURATION	-0.0
$LESS_THAN_DURATION_BASE_FIRST	-200_000.0
$LINE_TERMINATOR	ASCII.LF
$LOW_PRIORITY	0
$MACHINE_CODE_STATEMENT	NULL;
$MACHINE_CODE_TYPE	NO_SUCH_TYPE
$MANTISSA_DOC	31
$MAX_DIGITS	15
$MAX_INT	2147483647
$MAX_INT_PLUS_1	2_147_483_648
$MIN_INT	-2147483648
$NAME	SHORT_SHORT_INTEGER
$NAME_LIST	MIPS_BARE
$NAME_SPECIFICATION1	X2120A
$NAME_SPECIFICATION2	X2120B

$NAME_SPECIFICATION3	X3119A
$NEG_BASED_INT	16#FFFFFFFE#
$NEW_MEM_SIZE	2147483648
$NEW_SYS_NAME	MIPS_BARE
$PAGE_TERMINATOR	' '
$RECORD_DEFINITION	NEW INTEGER
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	10240
$TICK	2.0 ** (-14)
$VARIABLE_ADDRESS	GET_VARIABLE_ADDRESS
$VARIABLE_ADDRESS1	GET_VARIABLE_ADDRESS1
$VARIABLE_ADDRESS2	GET_VARIABLE_ADDRESS2

APPENDIX B

COMPILATION AND LINKER SYSTEM OPTIONS

The compiler and linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

4 Compiling

After a program library has been created, one or more compilation units can be compiled in the context of this library. The compilation units can be placed on different source files or they can all be on the same file. One unit, a parameterless procedure, acts as the main program. If all units needed by the main program and the main program itself have been compiled successfully, they can be linked. The resulting code can then be executed.

§4.1 and Chapter 5 describe in detail how to call the Compiler, the Completer, which is called to generate code for instances of generic units, and the Linker.
Chapter 6 explains the information which is given if the execution of a program is abandoned due to an unhandled exception.
The information the Compiler produces and outputs in the Compiler listing is explained in §4.4.
Finally, the log of a sample session is given in Chapter 7.

4.1 Compiling Ada Units

The command `xada.c` invokes the Compiler, and optionally Completer and Linker of the Alsys Ada System.

xada.c — Command Description

NAME

xada.c – Alsys Ada System compile command

SYNOPSIS

xada.c [option ...] [file ...]

DESCRIPTION

Compilation, Completion and Linking are performed in that order. The Completer is called if the `-C` or the `-m` option is specified. The Linker is called if the `-m` option is specified. By default, only the compiler runs and compiles the source(s) in the given *files*.

By default, the normal, full compilation is done.

`-B` *file* The option `-B` specifies a collection image file (a file containing the result of a previous incremental link). If a base is specified, then the final link is done on the base of the given file.

`-C` *unitlist* Requests the completion of the units in *unitlist*, which is a white space separated list of unit names. *unitlist* must be a single shell argument and must therefore be quoted when it has more than one item. Example with two units:

```
   xada.c -C "our_unit my_unit"
```

The Completer generates code for all instantiations of generic units in the execution closure of the specified unit(s). It also generates code for packages without bodies (if necessary).
If a listing is requested the default filename used is `complete.l`. The listing file contains the listing information for all units given in *unitlist*.

`-c` Controls whether a copy of the source file is kept in the library. The copy in the program library is used for later access by the Debugger or tools like the Recompiler. The name of the copy is generated by the Compiler and need normally not be known by the user. The Recompiler and the Debugger know this name. You can use the `xada.list -l` command to see the file name of the copy. If a specified file contains several compilation units a copy containing only the source text of one compilation unit is stored in the library for each compilation unit. Thus the Recompiler can recompile a single unit.

If `-c` is specified, the Compiler only stores the name of the source file in the program library. In this case the Recompiler and the Debugger are able to use the original file if it still exists.

`-D` When linking, the generation of debug information is suppressed.

`-d` *file* Specifies the name of the file which contains the linker directives. They describe the target's memory regions and prescribe the mapping of code, data, stack, and heap sections into these regions. For its format see §5.3. The Alsys Ada System is delivered with two directive files for the IDT 7RS301. Each of them can be used for those applications that use the whole memory of the IDT 7RS301 and do not require specific units to be linked into specific regions. The difference between them is that the file `<ADA_dir>/idt_min.lid` must be used when the main part of the Minimal Target Kernel is given with the option `-e`, and the file `<ADA_dir>/idt_full.lid` can be used when it is not given. If you want to use one of these files, specify:

`-M` *file*	Specifies whether the map listing of the Linker and the table of symbols which are used for linking the Ada units are to be produced in the specified file.
`-m` *unit*	Specifies the name of a main program, which must be a parameterless procedure. This option will cause the completion of any generic instantiations in the program; if a listing is requested, the listing options have the same meaning as for the complete option; if the completer has already been called by the `-C` option, the listing output is appended to that completer listing file. If all compilations are successful, the linker is invoked to build an program image; if a listing is requested, the default filename for the linker listing is `link.l`.
`-O`*l*	Restricts optimizations to level *l*. Level 0 indicates no optimizations, level 1 indicates partial optimizations, level 2 indicates full optimization. Default is full optimization. Partial optimizations allows those optimizations that do not move code globally. These are: Constant propagation, copy propagation, algebraic simplifications, runtime check elimination, dead code elimination, peephole and pipeline optimizations. This optimization level allows easier debugging while maintaining a reasonable code quality. Full optimization enforces the following optimizations in addition to those done with `-O1`: Global common subexpression elimination and keeping local variables in registers.
`-o` *file*	When linking is requested by `-m` this option can be used to specify the name of the program image file. By default, the unit name given with the `-m` option is used with the suffix .e. The given unit name is taken literally, i.e. upper and lower case letters are distinguished.
`-R`	Indicates that a recompilation of a previously analyzed source is to be performed. This option should only be used in commands produced by the `xada.make` command.
`-S`	Controls whether all run-time checks are suppressed. If you specify `-S` this is equivalent to the use of `PRAGMA suppress` for all kinds of checks. By default, no run-time checks are suppressed, except in cases where `PRAGMA suppress_all` appears in the source.
`-s`	Controls whether machine code is appended to the listing file. `-s` has no effect if no listing is requested or `-a` (analyze only) is specified. By default, no machine code is appended to the listing file.

5 Linking

The Linker of the Alsys Ada System either performs incremental linking or final linking.

Final linking produces a *program image file* which contains a loadable program. The *code portion* which is part of the program image file must be loaded onto the target later on. Final linking can (but need not) be based on the results of previous incremental linking.

Incremental linking means that a program is linked step by step (say in $N \geq 1$ steps 1 ... N). All steps except the last one are called incremental linking steps. In an incremental linking step, a *collection image file* containing a *collection* is produced; a collection is a set of Ada units and external units.

Each step $X \in \{2 \dots N\}$ is based on the result of step X-1. The last step is always a final link, i.e. it links the Ada main program. The result of an incremental linking step is also a code portion which must be loaded onto the target later on.

So the code of a program may consist of several code portions which are loaded onto the target one by one. This is called *incremental loading.*

The reasons for the introduction of the concept of incremental linking and loading into the Ada Cross System are the following:

- It should be possible that some Ada library units and external units are compiled, linked, and burnt into a ROM that is plugged into the target, and that programs using these units are linked afterwards.
- The loading time during program development should be as short as possible. This is achieved by linking those parts of the program that are not expected to be changed (e.g. some library units and the Ada Runtime System). The resulting code portion is loaded to the target and need not be linked or loaded later on. Instead, only those parts of the program that have been modified or introduced since the first link must be linked, so that the resulting code portion is much smaller in size than the code of the whole program would be. Because typically this code portion is loaded several times during program development, the development cycle time is reduced drastically.

The Runtime System (which is always necessary for the execution of Ada programs) is always linked during the first linking step. In particular, this means that also the version of the Runtime System (Debug or Non-Debug) is fixed during the first step.

The Linker gives the user great flexibility by allowing him to prescribe the mapping of single Ada units and assembler routines into the memory of the target. This, for

xada.link — Command Description

NAME

xada.link – final link of an Ada program

SYNOPSIS

xada.link [option ...] unit

DESCRIPTION

The `xada.link` command invokes the Alsys Ada Linker for final linking.

The Linker generates a program image file, the code portion of which can be loaded by the `xada.load` command and executed by the `xada.start` command. The default file name of the program image file is the unit name of the main program with suffix .e. The unit name given as parameter is taken literally, i.e. upper and lower case letters are distinguished.

unit specifies the library unit which is the main program. This must be a parameterless library procedure.

`-A`	This option is passed to the implicitly invoked Completer. See the same option with the `xada.c` command.
`-B` *file*	The option `-B` specifies a collection image file (a file containing the result of a previous incremental link). If a base is specified, then the final link is done on the base of the given file.
`-c`	Suppresses invokation of the Completer of the Alsys Ada System before the linking is performed. Only specify `-c` if you are sure that there are no instantiations or implicit package bodies to be compiled, e.g. if you repeat the `xada.link` command with different linker options.
`-D`	By default debug information for the Alsys Ada Debugger is generated and included in the program image file. When the `-D` option is present, debug information is not included in the program image file. If the program is to run under the control of the Debugger it must be linked without the `-D` option.

	To run a program under the control of the Full Target Kernel, the option `-k` is must not be specified.
`-l`	If `-l` is specified the Linker of the Alsys Ada System creates a listing file containing a table of symbols which are used for linking the Ada units. This table is helpful when debugging an Ada program with the Target Board PROM debugger. The default name of the listing file is `link.l`. By default, the Linker does not create a listing file. This option is also passed to the implicitly invoked Completer, which by default generates a listing file `complete.l` if `-l` is given.
`-L` *directory*	The listing files are created in directory *directory* instead of in the current directory (default).
`-L` *file*	The listing files are concatenated onto file *file*.
`-M` *file*	Specifies whether the map listing of the Linker and the table of symbols which are used for linking the Ada units are to be produced in the specified file.
`-O`*l*	This option is passed to the implicitly invoked Completer. See the same option with the `xada.c` command.
`-o` *file*	Specifies the name of the image file. The default file name of the program image file is the unit name of the main program with suffix .e. The given unit name is taken literally, i.e. upper and lower case letters are distinguished.
`-S`	This option is passed to the implicitly invoked Completer. See the same option with the `xada.c` command.
`-s`	This option is passed to the implicitly invoked Completer. See the same option with the `xada.c` command. If a listing is requested and `-s` is specified, the Linker of the Alsys Ada System generates a listing with the machine code of the program starter in the file `link.l`. The program starter is a routine which contains the calls of the necessary elaboration routines and a call for the Ada subprogram which is the main program. By default, no machine code is generated.
`-v`	Controls whether the `xada.link` command writes additional information onto standard error, and is also passed to the implicitly invoked Completer. By default, no additional information is written.
`-y` *library*	Specifies the program library the command works on. The `xada.link` command needs write access to the library unless `-c`

This program image file serves as input for the Loader or the Debugger in order to load the code portion included in the file onto the target, or for the Starter or the Debugger in order to start the linked program, or for the Format Converter.

5.2 Linking Collections

A set of Ada units and external units which can be linked separately is called a collection. Such a collection consists on one hand of all compilation units needed by any of the given library units, and on the other hand of all given external units. All compilation units must successfully have been compiled or completed previously.

The code of a linked collection does not contain any unresolved references and can thus be loaded to the target and used by programs linked afterwards without any changes. In particular, this allows the code of a linked collection to be burnt into a ROM. Linking a collection is called incremental linking.

Contrary to final linking, incremental linking is not done selectively. Instead all code and data belonging to the collection is linked, because the Linker does not know which programs or collections will be linked on the collection as a base.

Incremental linking results in a collection image file. There is a code portion in this image file which, together with the code of the given base (if any), is the code of all Ada units and all external (assembler written) units that belong to the collection.

For incremental linking, the Linker is started by the `xada.ilink` command.

xada.ilink — Command Description

NAME

xada.ilink – incremental link of an Ada program

SYNOPSIS

xada.ilink [option ...] file

DESCRIPTION

The `xada.ilink` command invokes the Alsys Ada Linker for incremental linking.

-u *unitlist*	Specifies a list of Ada library units that are to be linked. A unit of the list denotes a library unit within the given program library. All specified library units together with their secondary units and all units needed by them must have been successfully compiled and completed (cf. Chapter 4).
-v	Controls whether the `xada.link` command writes additional information onto standard error, and is also passed to the implicitly invoked Completer. By default, no additional information is written.
-y *library*	Specifies the program library the command works on.

End of Command Description

The option -u and the option -e define a collection C as defined at the beginning of this section. If a base collection is specified with the option -B, then C is enlarged by all units belonging to this base collection. The units belonging to the base collection are identified by their names (Ada name of a library unit or name of the external unit) and by their compilation or assembly times. The Linker uses these to check whether a base unit is obsolete or not.

See §5.4 for the mapping process.

If no errors are detected within the linking process, then the result of an incremental link is a collection image file containing the following:

- A code portion that contains the complete code of the linked collection, except the code of the base collection.
- Base addresses and lengths of the regions actually occupied by the complete collection (including the base collection).
- Checksums of the regions which contain code sections and which are actually occupied by the complete collection (including the base collection).
- The names of all library units as specified by the user with the option -u (including those of the base collection).
- The list of all compilation units and external units that belong to the complete linked collection (including the base collection), together with their compilation (resp. assembly) times.
- Information about all sections belonging to the complete linked collection (including the base collection), and about the symbols which they define and refer.

This collection image file serves as input for the Loader or the Debugger in order to load the code portion included in the file onto the target, or for the Linker as a base collection file, or for the Format Converter.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are contained in the following Predefined Language Enviroment (chapter 13 page 287 ff of the compiler user manual).

13 Predefined Language Environment

The predefined language environment comprises the package standard, the language-defined library units and the implementation-defined library units.

13.1 The Package STANDARD

The specification of the package standard is outlined here; it contains all predefined identifiers of the implementation.

```
PACKAGE standard IS

  TYPE boolean IS (false, true);

  -- The predefined relational operators for this type are as follows:

  -- FUNCTION "="   (left, right : boolean) RETURN boolean;
  -- FUNCTION "/="  (left, right : boolean) RETURN boolean;
  -- FUNCTION "<"   (left, right : boolean) RETURN boolean;
  -- FUNCTION "<="  (left, right : boolean) RETURN boolean;
  -- FUNCTION ">"   (left, right : boolean) RETURN boolean;
  -- FUNCTION ">="  (left, right : boolean) RETURN boolean;

  -- The predefined logical operators and the predefined logical
  -- negation operator are as follows:

  -- FUNCTION "AND" (left, right : boolean) RETURN boolean;
  -- FUNCTION "OR"  (left, right : boolean) RETURN boolean;
  -- FUNCTION "XOR" (left, right : boolean) RETURN boolean;
  -- FUNCTION "NOT" (right : boolean) RETURN boolean;

  -- The universal type universal_integer is predefined.

  TYPE integer IS RANGE - 2_147_483_648 .. 2_147_483_647;

  -- The predefined operators for this type are as follows:

  -- FUNCTION "="  (left, right : integer) RETURN boolean;
  -- FUNCTION "/=" (left, right : integer) RETURN boolean;
  -- FUNCTION "<"  (left, right : integer) RETURN boolean;
-- FUNCTION "+"   (left, right : float) RETURN float;
-- FUNCTION "-"   (left, right : float) RETURN float;
-- FUNCTION "*"   (left, right : float) RETURN float;
-- FUNCTION "/"   (left, right : float) RETURN float;

-- FUNCTION "**"  (left : float; right : integer) RETURN float;

-- An implementation may provide additional predefined floating
-- point types. It is recommended that the names of such additional
-- types end with FLOAT as in SHORT_FLOAT or LONG_FLOAT.
-- The specification of each operator for the type  universal_real,
-- or for any additional predefined floating point type, is obtained
-- by replacing FLOAT by the name of the type in the specification of
-- the corresponding operator of the type FLOAT.

TYPE long_float IS DIGITS 15 RANGE
    - 16#0.FFFF_FFFF_FFFF_E#E256 .. 16#0.FFFF_FFFF_FFFF_E#E256;

-- In addition, the following operators are predefined for universal
-- types:

-- FUNCTION "*" (left : UNIVERSAL_INTEGER; right : UNIVERSAL_REAL)
                RETURN UNIVERSAL_REAL;
-- FUNCTION "*" (left : UNIVERSAL_REAL;    right : UNIVERSAL_INTEGER)
                RETURN UNIVERSAL_REAL;
-- FUNCTION "/" (left : UNIVERSAL_REAL;    right : UNIVERSAL_INTEGER)
                RETURN UNIVERSAL_REAL;

-- The type universal_fixed is predefined.
-- The only operators declared for this type are

-- FUNCTION "*" (left : ANY_FIXED_POINT_TYPE;
                right : ANY_FIXED_POINT_TYPE) RETURN UNIVERSAL_FIXED;
-- FUNCTION "/" (left : ANY_FIXED_POINT_TYPE;
                right : ANY_FIXED_POINT_TYPE) RETURN UNIVERSAL_FIXED;

-- The following characters form the standard ASCII character set.
-- Character  literals corresponding to control characters are not
-- identifiers.

TYPE character IS
     (nul,  soh,  stx,  etx,     eot,  enq,  ack,  bel,
      bs,   ht,   lf,   vt,      ff,   cr,   so,   si,
      dle,  dc1,  dc2,  dc3,     dc4,  nak,  syn,  etb,
      can,  em,   sub,  esc,     fs,   gs,   rs,   us,
      ' ',  '!',  '"',  '#',     '$',  '%',  '&',  ''',
    semicolon  : CONSTANT character := ';';
    query      : CONSTANT character := '?';
    at_sign    : CONSTANT character := '@';
    l_bracket  : CONSTANT character := '[';
    back_slash : CONSTANT character := '\';
    r_bracket  : CONSTANT character := ']';
    circumflex : CONSTANT character := '^';
    underline  : CONSTANT character := '_';
    grave      : CONSTANT character := '`';
    l_brace    : CONSTANT character := '{';
    bar        : CONSTANT character := '|';
    r_brace    : CONSTANT character := '}';
    tilde      : CONSTANT character := '~';

    lc_a : CONSTANT character := 'a';
    ...
    lc_z : CONSTANT character := 'z';

  END ascii;

  -- Predefined subtypes:

  SUBTYPE natural  IS integer RANGE 0  . integer'last;
  SUBTYPE positive IS integer RANGE 1 .. integer'last;

  -- Predefined string type:

  TYPE string IS ARRAY(positive RANGE <>) OF character;
  PRAGMA pack(string);

  -- The predefined operators for this type are as follows:

  -- FUNCTION "="  (left, right : string) RETURN boolean;
  -- FUNCTION "/=" (left, right : string) RETURN boolean;
  -- FUNCTION "<"  (left, right : string) RETURN boolean;
  -- FUNCTION "<=" (left, right : string) RETURN boolean;
  -- FUNCTION ">"  (left, right : string) RETURN boolean;
  -- FUNCTION ">=" (left, right : string) RETURN boolean;

  -- FUNCTION "&" (left : string;    right : string)    RETURN string;
  -- FUNCTION "&" (left : character; right : string)    RETURN string;
  -- FUNCTION "&" (left : string;    right : character) RETURN string;
  -- FUNCTION "&" (left : character; right : character) RETURN string;

  TYPE duration IS DELTA 2#1.0#E-14 RANGE
          - 131_072.0 .. 131_071.999_938_964_843_75;
```

13.3.1 The Package COLLECTION_MANAGER

In addition to unchecked storage deallocation (cf. LRM §13.10.1), this implementation provides the generic package `collection_manager`, which has advantages over unchecked deallocation in some applications; e.g. it makes it possible to clear a collection with a single reset operation. See §15.10 for further information on the use of the collection manager and unchecked deallocation.

The package specification is:

```
GENERIC
   TYPE elem IS LIMITED PRIVATE;
   TYPE acc  IS ACCESS elem;
PACKAGE collection_manager IS

   TYPE status IS LIMITED PRIVATE;

   PROCEDURE mark (s : OUT status);

      -- Marks the heap of type ACC and
      -- delivers the actual status of this heap.

   PROCEDURE release (s : IN status);

      -- Restore the status s on the collection of ACC.
      -- RELEASE without previous MARK raises CONSTRAINT_ERROR

   PROCEDURE reset;

      -- Deallocate all objects on the heap of ACC

   PRIVATE

      -- private declarations

END collection_manager;
```

A call of the procedure `release` with an actual parameter `s` causes the storage occupied by those objects of type `acc` which were allocated after the call of `mark` that delivered `s` as result, to be reclaimed. A call of `reset` causes the storage occupied by all objects of type `acc` which have been allocated so far to be reclaimed and cancels the effect of all previous calls of `mark`.

```
WITH system;
WITH calendar;

PACKAGE privileged_operations IS

   SUBTYPE word_range      IS integer RANGE -2**31 .. 2**31-1;
   SUBTYPE half_word_range IS integer RANGE -2**15 .. 2**15-1;
   SUBTYPE byte_range      IS integer RANGE -2**7  .. 2**7-1;

   SUBTYPE bit_number IS integer RANGE 0 .. 7;
      -- 0 designates the least significant bit of a byte.
      -- 7 designates the most significant bit of a byte.

   SUBTYPE interrupt_number IS integer RANGE 0 .. 5;

   PROCEDURE assign_byte (dest : system.address;
                          item : byte_range);
   PROCEDURE assign_half_word (dest : system.address;
                               item : half_word_range);
   PROCEDURE assign_word (dest : system.address;
                          item : word_range);
   PROCEDURE assign_addr (dest : system.address;
                          item : system.address);

   PROCEDURE bit_set   (dest  : system.address;
                        bitno : bit_number);
   PROCEDURE bit_clear (dest  : system.address;
                        bitno : bit_number);

   FUNCTION byte_value (addr : system.address) RETURN byte_range;
   FUNCTION half_word_value
                       (addr : system.address) RETURN half_word_range;
   FUNCTION word_value (addr : system.address) RETURN word_range;
   FUNCTION addr_value (addr : system.address) RETURN system.address;

   FUNCTION bit_value (addr  : system.address;
                       bitno : bit_number) RETURN boolean;
      -- true is returned if the bit is set, false otherwise.

   PROCEDURE define_interrupt_service_routine
                (routine       : system.address;
                 for_interrupt : interrupt_number);
      -- defines an assembler routine as interrupt service routine

   PROCEDURE set_date_time (now : calendar.time);

END privileged_operations;
```

15 Appendix F

This chapter, together with the Chapters 16 and 17, is the Appendix F required in the LRM, in which all implementation-dependent characteristics of an Ada implementation are described.

15.1 Implementation-Dependent Pragmas

The form, allowed places, and effect of every implementation-dependent pragma is stated in this section.

15.1.1 Predefined Language Pragmas

The form and allowed places of the following pragmas are defined by the language; their effect is (at least partly) implementation-dependent and stated here.

CONTROLLED
: has no effect.

ELABORATE
: is fully implemented. The Alsys Ada System assumes a PRAGMA elaborate, i.e. stores a unit in the library as if a PRAGMA elaborate for a unit u was given, if the compiled unit contains an instantiation of u (or a generic program unit in u) and if it is clear that u *must* have been elaborated before the compiled unit. In this case an appropriate information message is given. By this means it is avoided that an elaboration order is chosen which would lead to a PROGRAM_ERROR when elaborating the instantiation.

INLINE
: Inline expansion of subprograms is supported with the following restrictions: the subprogram must not contain declarations of other subprograms, tasks, generic units or body stubs. If the subprogram is called recursively only the outer call of this subprogram will be expanded.

SHARED
is fully supported.

STORAGE_UNIT
has no effect.

SUPPRESS
has no effect, but see §15.1.2 for the implementation-defined PRAGMA `suppress_all`.

SYSTEM_NAME
has no effect.

15.1.2 Implementation-Defined Pragmas

BYTE_PACK
see §16.1.

EXTERNAL_NAME (<string>, <ada_name>)
<ada_name> specifies the name of a subprogram or of an object declared in a library package, <string> must be a string literal. It defines the external name of the specified item.
This pragma is used in connection with PRAGMA `interface` (see §15.1.1). If <ada_name> is the name of a subprogram, the Compiler uses the symbol <string> in the `call` instruction for the subprogram. The subprogram declaration of <ada_name> must precede this pragma. If several subprograms with the same name satisfy this requirement the pragma refers to that subprogram which is declared last.
If <ada_name> is the name of an object, this pragma enables the object to be accessed from outside the Ada program using the symbol <string>, for example from a subprogram written in another language.
Upper and lower cases are distinguished within <string>, i.e. <string> must be given exactly as it is to be used by external routines.

If this procedure is compiled by the Alsys Ada Compiler without suppression of dead code elimination, i.e. with the `-O0` option, the first assignment to `fcb` will be eliminated, because the Compiler will not recognize that the value of `fcb` may be read before the next assignment to `fcb`. Therefore

```
PRAGMA resident (fcb);
```

should be inserted after the declaration of `fcb`.

This pragma can be applied to all those kinds of objects for which the address clause is supported (cf. §16.5).

SUPPRESS_ALL
causes all the runtime checks described in the LRM §11.7 to be suppressed; this pragma is only allowed at the start of a compilation before the first compilation unit; it applies to the whole compilation.

15.1.3 Pragma Interface (Assembler,...)

This section describes the internal calling conventions of the Alsys Ada System, which are the same ones which are used for subprograms for which a `PRAGMA interface (ASSEMBLER,...)` is given. Thus the actual meaning of this pragma is simply that the body needs and must not be provided in Ada; it is provided in object form using the `-e` option with the `xada.link` (or `xada.c` or `xada.make`) command.

The internal calling conventions are explained in four steps:

- Parameter passing mechanism
- Ordering of parameters
- Type mapping
- Saving registers

Parameter passing mechanism:

The Alsys Ada System uses three different parameter passing mechanisms, depending on the type of a parameter:

- *by value and/or result*: The value of the parameter itself is passed.

in the array (i.e. `FIRST(1)`, `LAST(1)`, `FIRST(2)`, `LAST(2)`, ...). The space allocated for the bound elements in the descriptor depends on the type of the index constraint. This descriptor is itself passed by reference.

- For functions whose return value is an unconstrained array type, a reference to a descriptor for the array is passed in the parameter block as for parameters of mode `OUT`. The fields for its address and all array index bounds are filled up by the function before it returns. In contrast to the procedure for an `OUT` parameter, the function allocates the array in its own stack space. The function then returns without releasing its stack space. After the function has returned, the calling routine copies the array into its own memory space and then deallocates the stack memory of the function.
- A constrained record parameter is passed by reference for all parameter modes.
- For an unconstrained record parameter of mode `IN`, the parameter is passed by reference using the address pointing to the record. If the parameter has mode `OUT` or `IN OUT`, the value of the `CONSTRAINED` attribute applied to the actual parameter is passed as an additional boolean `IN` parameter (which, when not passed in a register, occupies one byte in the parameter block and is aligned to a byte boundary). The boolean `IN` parameter and the address are treated like two consecutive parameters in a subprogram specification, i.e. the positions of the two parameters within the parameter block are determined independently of each other.

For all kinds of composite parameter types, the pointer pointing to the actual parameter object is represented by a 32 bit address, which is always aligned to a word boundary.

Ordering of parameters:

The ordering of the parameters is determined as follows:

The parameters are processed in the order they are defined in the Ada subprogram specification. For a function, the return value is treated as an anonymous parameter of mode `OUT` at the start of the parameter list. The registers `$4..$22` and `$f0..$f31` are available for parameter passing. A parameter block is only used when there are more parameters than registers of the appropriate class. Registers are used from low numbers to high numbers, the parameter block starts at offset zero and grows to higher offsets. Each parameter is handled as follows:

- A `float` parameter is allocated the next free even numbered floating point register (the corresponding odd numbered floating point register is not used for parameter passing). If there is no free floating point register, one word is allocated in the parameter block (see below).
- A `long_float` parameter is allocated the next free floating point register pair. If there is no free floating point register pair, a double word is allocated in the parameter block (see below).

to a byte, word, longword or a multiple thereof depending on the size and alignment requirements of the components' subtype. This padding may be influenced using one of the PRAGMAs pack or byte_pack (cf. §16.1). The offset of an individual array component is then obtained by multiplying the padded size of one array component by the number of components stored in the array before it. This number may be determined from the number of elements for each dimension using the fact that the array elements are stored row by row. (For unconstrained arrays the number of elements for each dimension can be found in the descriptor, which is passed by reference.)

A record object is implemented as a concatenation of its components. Initially, locations are reserved for those components that have a component clause applied to them. Then locations for all other components are reserved. Any gaps large enough to hold components without component clauses are filled, so in general the record components are rearranged. Components in record variants are overlaid. The ordering mechanism of the components within a record is in principle the same as that for ordering the parameters in the parameter block.

A record may hold implementation-dependent components (cf. §16.4). For a record component whose size depends on discriminants, a generated component holds the offset of the record component within the record object. If a record type includes variant parts there may be a generated component (cf. §16.4) holding the size of the record object. This size component is allocated as the first component within the record object if this location is not reserved by a component clause. Since the mapping of record types is rather complex record component clauses should be introduced for each record component if an object of that type is to be passed to a non Ada subprogram to be sure to access the components correctly.

Saving registers:

The last aspect of the calling conventions discussed here is that of saving registers. The calling subprogram assumes that the values of the registers $1..$22, $24..$25 will be destroyed by the called subprogram, and therefore saves them of its own accord. The stack pointer $29 will have the same value after the call as before except for functions returning unconstrained arrays. The stack limit register ($23) will have the same value after the call as before unless the stack of the main task was extended. If the called subprogram wants to modify further registers it has to ensure that the old values are restored upon return from the subprogram. Note that these register saving conventions differ from the calling standard described in [R3000].

Finally we give the appropriate code sequences for the subprogram entry and for the return, which both obey the rules stated above.

A subprogram for which PRAGMA interface (assembler,...) is specified is - in effect - called with the subprogram calling sequence

15.2.1 Language-Defined Attributes

The name and type of all the language-defined attributes are as given in the LRM. We note here only the implementation-dependent aspects.

ADDRESS

If this attribute is applied to an object for which storage is allocated, it yields the address of the first storage unit that is occupied by the object.
If it is applied to a subprogram or to a task, it yields the address of the entry point of the subprogram or task body.
If it is applied to a task entry for which an address clause is given, it yields the address given in the address clause.
For any other entity this attribute is not supported and will return the value `system.address_zero`.

IMAGE

The image of a character other than a graphic character (cf. LRM §3.5.5(11)) is the string obtained by replacing each italic character in the indication of the character literal (given in the LRM Annex C(13)) by the corresponding upper-case character. For example, `character'image(`*nul*`) = "NUL"`.

MACHINE_OVERFLOWS

Yields `true` for each real type or subtype.

MACHINE_ROUNDS

Yields `true` for each real type or subtype.

STORAGE_SIZE

The value delivered by this attribute applied to an access type is as follows:
If a length specification (`STORAGE_SIZE`, see §16.2) has been given for that type (static collection), the attribute delivers that specified value.
In case of a dynamic collection, i.e. no length specification by `STORAGE_SIZE` given for the access type, the attribute delivers the number of storage units currently allocated for the collection. Note that dynamic collections are extended if needed.
If the collection manager (cf. §13.3.1) is used for a dynamic collection the attribute delivers the number of storage units currently allocated for the collection. Note that in this case the number of storage units currently allocated may be decreased by release operations.

```
   -- than '0'..'9', 'a'..'f', 'A'..'F' or if the resulting address
   -- value cannot be represented with 32 bits.

FUNCTION convert_address (addr : address) RETURN external_address;
   -- The resulting external address consists of exactly 8
   -- characters '0'..'9', 'A'..'F'.

TYPE name IS (mips_bare);
system_name  : CONSTANT name := mips_bare;

storage_unit : CONSTANT := 8;
memory_size  : CONSTANT := 2 ** 31;
min_int      : CONSTANT := - 2 ** 31;
max_int      : CONSTANT := 2 ** 31 - 1;
max_digits   : CONSTANT := 15;
max_mantissa : CONSTANT := 31;
fine_delta   : CONSTANT := 2.0 ** (-31);
tick         : CONSTANT := 2.0 ** (-14);

SUBTYPE priority IS integer RANGE 0 .. 15;

TYPE interrupt_number IS RANGE 1 .. 32;
   -- User defined interrupts

interrupt_vector : ARRAY (interrupt_number) OF address;
   -- converts an interrupt_number to an address;

non_ada_error : EXCEPTION RENAMES _non_ada_error;
   -- non_ada_error is raised, if some event occurs which does not
   -- correspond to any situation covered by Ada, e.g.:
   --    illegal instruction encountered
   --    error during address translation
   --    illegal address

TYPE exception_id IS NEW address;

no_exception_id     : CONSTANT exception_id := NULL;

-- Coding of the predefined exceptions:

FUNCTION constraint_error_id RETURN exception_id;
FUNCTION numeric_error_id    RETURN exception_id;
FUNCTION program_error_id    RETURN exception_id;
FUNCTION storage_error_id    RETURN exception_id;
FUNCTION tasking_error_id    RETURN exception_id;
FUNCTION non_ada_error_id    RETURN exception_id;
FUNCTION status_error_id     RETURN exception_id;
```

```
   TYPE exit_code IS NEW integer;

   error        : CONSTANT exit_code := 1;
   success      : CONSTANT exit_code := 0;

   PROCEDURE set_exit_code (val : exit_code);
      -- Specifies the exit code which is returned to the
      -- operating system if the Ada program terminates normally.
      -- The default exit code is 'success'. If the program is
      -- abandoned because of an exception, the exit code is
      -- 'error'.

PRIVATE

   -- private declarations

END system;
```

15.4 Restrictions on Representation Clauses

See Chapter 16 of this manual.

15.5 Conventions for Implementation-Generated Names

There are implementation generated components but these have no names. (cf. §16.4 of this manual).

15.6 Expressions in Address Clauses

See §16.5 of this manual.

15.11 Machine Code Insertions

A package machine_code is not provided and machine code insertions are not supported.

15.12 Numeric Error

The predefined exception numeric_error is never raised implicitly by any predefined operation; instead the predefined exception constraint_error is raised.

16 Appendix F: Representation Clauses

In this chapter we follow the section numbering of Chapter 13 of the LRM and provide notes for the use of the features described in each section.

16.1 Pragmas

PACK

As stipulated in the LRM §13.1, this pragma may be given for a record or array type. It causes the Compiler to select a representation for this type such that gaps between the storage areas allocated to consecutive components are minimized. For components whose type is an array or record type the PRAGMA PACK has no effect on the mapping of the component type. For all other component types the Compiler will choose a representation for the component type that needs minimal storage space (packing down to the bit level). Thus the components of a packed data structure will in general not start at storage unit boundaries.

BYTE_PACK

This is an implementation-defined pragma which takes the same argument as the predefined language PRAGMA PACK and is allowed at the same positions. For components whose type is an array or record type the PRAGMA BYTE_PACK has no effect on the mapping of the component type. For all other component types the Compiler will try to choose a more compact representation for the component type. But in contrast to PRAGMA PACK all components of a packed data structure will start at storage unit boundaries and the size of the components will be a multiple of `system.storage_unit`. Thus, the PRAGMA BYTE_PACK does not effect packing down to the bit level (for this see PRAGMA PACK).

16.4 Record Representation Clauses

Record representation clauses are supported. The value of the expression given in an alignment clause must be 0, 1, 2 or 4. If this restriction is violated, the Compiler responds with a RESTRICTION error message in the Compiler listing. If the value is 0 the objects of the corresponding record type will not be aligned, if it is 1, 2 or 4 the starting address of an object will be a multiple of the specified alignment.

The number of bits specified by the range of a component clause must not be greater than the amount of storage occupied by this component. (Gaps between components can be forced by leaving some bits unused but not by specifying a bigger range than needed.) Violation of this restriction will produce a RESTRICTION error message.

There are implementation-dependent components of record types generated in the following cases :

- If the record type includes variant parts and the difference between the maximum and the minimum sizes of the variant is greater than 32 bytes, and, in addition, if it has either more than one discriminant or else the only discriminant may hold more than 256 different values, the generated component holds the size of the record object. (If the second condition is not fulfilled, the number of bits allocated for any object of the record type will be the value delivered by the size attribute applied to the record type.)
- If the record type includes array or record components whose sizes depend on discriminants, the generated components hold the offsets of these record components (relative to the corresponding generated component) in the record object.

But there are no implementation-generated names (cf. LRM §13.4(8)) denoting these components. So the mapping of these components cannot be influenced by a representation clause.

16.5 Address Clauses

Address clauses are supported for objects declared by an object declaration and for single task entries. If an address clause is given for a subprogram, package or a task unit, the Compiler responds with a RESTRICTION error message in the Compiler listing.

If an address clause is given for an object, the storage occupied by the object starts at the given address. Address clauses for single entries are described in §16.5.1.

for each logical interrupt number (index in system.interrupt_vector). The main purpose of an ISR is to set the corresponding boolean flag of the variable _IRENTRYC to true to indicate the occurrence of the logical interrupt to the Ada runtime system. The range 1 .. 32 of the type system.interrupt_number corresponds to bit 0 .. 31 of _IRENTRYC. After the return of the ISR to the Target Kernel, the Target Kernel inspects the boolean array and, if at least one flag is set, gives control to the Ada runtime system. The Ada runtime system checks _IRENTRYC, identifies the logical interrupt number and executes the corresponding entry call.

Before the ISR gives control back to the Target Kernel, the reason for the hardware interrupt has to be removed. This means that, for example, the ISR has to ensure that the corresponding bit in the Cause register has been cleared (e.g. reading a character from the DUART clears the corresponding bit in the Cause register). Otherwise the ISR will immediately be invoked again, which will cause an endless loop.

The ISR must not destroy any register except register $1.

3. Each occurrence of a hardware interrupt first affects the Target Kernel. In order to cause the ISR to be called whenever the corresponding hardware interrupt occurs, the Ada program must introduce the ISR to the Target Kernel by calling the procedure privileged_operations.define_interrupt_service_routine (see §13.3.3.). The first parameter of this procedure specifies the address of the ISR, and the second parameter identifies the hardware interrupt.

Both the initial interrupt handling within the Target Kernel and the user defined ISR run in (privileged) kernel mode of the processor. The Ada task which handles the interrupt runs in user mode.

Suppose, for example, that you want to catch the hardware interrupt 5. Then you must write an assembler routine, called ISR_5 in the following, which will be activated each time the interrupt 5 occurs (hardware interrupts have the number 0 .. 5). This routine is defined as an interrupt service routine by calling the PROCEDURE define_interrupt_service_routine of PACKAGE privileged_operations. In the Ada program, ISR_5 must be declared as an interrupt service routine as follows:

```
-- Declaration of the external routine ISR_5:
    PROCEDURE isr_5;
        PRAGMA interface (assembler, isr_5);
        PRAGMA external_name ("ISR_5", isr_5);

-- Definition of ISR_5 as interrupt service routine for interrupt 5:
    define_interrupt_service_routine (isr_5'address, 5);
```

The following scheme must be applied for an interrupt service routine :

```
PACKAGE terminal_global IS

   the_char : character;
      PRAGMA external_name ("THE_CHAR", the_char);

END terminal_global;

WITH privileged_operations,
     system,
     terminal_global,
     text_io;

USE  system;

PROCEDURE terminal_io_idt IS

   PRAGMA priority (2);

   PROCEDURE isr_read;
      PRAGMA interface (assembler, isr_read);
      PRAGMA external_name ("ISR_READ", isr_read);

   kseg1     : CONSTANT system.address :=
                  system.convert_address ("A0000000");
   dua_adr   : CONSTANT system.address := kseg1   + 16#1FE00000#;

   dua_imr   : CONSTANT system.address := dua_adr + 16#17#;
   dua_srb   : CONSTANT system.address := dua_adr + 16#27#;
   dua_thrb  : CONSTANT system.address := dua_adr + 16#2F#;

   TASK terminal_in IS
      PRAGMA priority (1);

      ENTRY char_entry;
      FOR char_entry USE AT system.interrupt_vector (1);
   END terminal_in;

   TASK terminal_out IS
      PRAGMA priority (0);

      ENTRY put (item : IN character);
   END terminal_out;

   TASK BODY terminal_in IS
      ch : character;
   BEGIN
```

```
 #-- CONSTANT SECTION
 #--
         .rdata
kseg1    =       0xA0000000
dua_adr  =       kseg1+0x1FE00000
dua_isr  =       dua_adr+0x17        #-- interrupt status register
 #--                                     port A
dua_rhrb =       dua_adr+0x2F        #-- rx holding register port B
 #--
         .text
 #--
 #---------------------------------------------------------------
         .ent    ISR_READ
ISR_READ:
 #-- Function:  Handler for RS232 Receive Interrupt from DUART (EI #5)
 #--            Handles only tty1 here
 #--
         sub     $sp,12
         .frame  $sp,12,$31
         .mask   -4,0x80000300
         sw      $8,0($sp)
         sw      $9,4($sp)
         sw      $31,8($sp)
 #--
         lbu     $8,dua_isr
         nop
 #--
         andi    $8,0x20             #-- test RxRDYB bit
         beq     $8,$0,no_char       #-- nothing from tty1
         nop
 #--
         lbu     $8,dua_rhrb         #-- read character into $8
         nop
 #--
 #-- clearing the pending interrupt and reset of the status bit in dua_isr

 #-- is implicitly done by reading dua_rhrb
 #--
         sb      $8,THE_CHAR         #-- make character available to
                                     #-- Ada prog.
                                     #-- make interrupt entry call
                                     #-- pending
         la      $8,_IRENTRYC
         lw      $9,0($8)
         nop
         ori     $9,1                #-- set logical interrupt #1 pending
```

17 Appendix F: Input-Output

In this chapter we follow the section numbering of Chapter 14 of the LRM and provide notes for the use of the features described in each section.

17.1 External Files and File Objects

The implementation only supports the files `standard_input` and `standard_output` of `PACKAGE text_io`. Any attempt to create or open a file raises the exception `use_error`.

17.2 Sequential and Direct Files

Sequential and direct files are not supported.

17.3 Text Input-Output

`standard_input` and `standard_output` are associated with the RS232 serial port of the target. If the Full Target Kernel is used, then all input/output operations are done on the host using the communication line between the host and this port. Both the Debugger and the Starter are prepared to do these I/O operations.

If the Minimal Target Kernel is used, then the same serial port as in the Full Target Kernel is used, but all data of `standard_output` is directly written to this port and all data of `standard_input` is directly read from this port.

For tasking aspects of I/O operations see Chapter 14.

For further details on the I/O implementation within the Target Kernel see Chapter 19.

17.3.1 Implementation-Defined Types

The implementation-dependent types `count` and `field` defined in the package specification of `text_io` have the following upper bounds:

```
PACKAGE low_level_io IS

   TYPE device_type IS (null_device);

   TYPE data_type IS
      RECORD
         NULL;
      END RECORD;

   PROCEDURE send_control    (device : device_type;
                              data   : IN OUT data_type);

   PROCEDURE receive_control (device : device_type;
                              data   : IN OUT data_type);

END low_level_io;
```

Note that the enumeration type `device_type` has only one enumeration value, `null_device`; thus the procedures `send_control` and `receive_control` can be called, but `send_control` will have no effect on any physical device and the value of the actual parameter data after a call of `receive_control` will have no physical significance.